MULTI-SELECTION IN TOUCH-SCREEN GRAPHICAL USER INTERFACES

A Thesis

by

ANIRUDH RAMANATHAN

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Jaakko Järvi
Committee Members,	Frank M. Shipman, III
	Kamran Entesari
Head of Department,	Dilma Da Silva

May 2016

Major Subject: Computer Science

ABSTRACT

Multi-selection is the act of selecting a set of elements on a graphical user interface in order to perform one or more operations on that set. Multi-selection is used, for example, to select thumbnails in an image gallery, in order to perform an action, such as uploading, deleting, or editing. This thesis builds on prior work that specifies multi-selection formally as a reusable feature for mouse-based interfaces. In this thesis we extend the formalism to touch-screen devices. The thesis also reports on *Multiselect-Android*, our implementation of the formalism for the Android platform.

The analysis and evaluation of the presented multi-selection framework were conducted in two stages. The first stage compares the implementation effort when a developer implements multi-selection using *Multiselect-Android* with the effort when using traditional vendor-provided GUI libraries. The second stage evaluates how easy and effective it is for users to select multiple elements using different kinds of multi-selection features. A user study that we conducted shows that multi-selection using Multiselect-Android is faster than using features that are common in today's applications. To summarize, this thesis defines generic selection semantics for touch-screen devices, describes a reference implementation, and compares the semantics with existing practice.

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my advisor Prof. Jaakko Järvi for the continuous support, patience, enthusiasm, and immense knowledge. His guidance helped me all through, from research methodology to theoretical concepts to carrying out the user study. The long meetings and discussions have taught me a lot and made me a better researcher. I am also grateful for his time and all the helpful comments I received during the writing of this thesis.

Besides my advisor, I would like to thank the rest of my advisory committee: Prof. Frank Shipman and Prof. Kamran Entesari, for their comments, and thought-provoking questions.

I would like to express my heartfelt thanks to my parents for their love and support throughout my life. To my friends and roommates: Ankur, Aprajith, Arun, Ashish, Divya, Gaurav, Komal, Monisha, Pallavi, Sushma, Vidhya and others, thank you for listening and supporting me throughout, and offering kind words during difficult times.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Mobile computing devices have become ubiquitous. Touch-screen interfaces are the de facto standard for interacting with these devices. These interfaces have evolved to support a variety of gestures, such as taps of different duration or multiplicities, swipes, contact with multiple fingers [4], and so forth. These gestures are building blocks for more complex forms of interaction. One common form, *multi-selection*, is used for selecting one or more items from a collection of items, in order to be able to apply an action to them. Examples of this interaction include the selection of thumbnails in a gallery application and file manipulation in a file-explorer application. Multi-selection is a frequently used feature and thus much work has gone into coming up with intuitive and convenient ways to perform multi-selection.

In the prominent mobile platforms of today, in order to facilitate application development and provide consistent behavior across applications, platform and third-party vendors package user interface components and make them available to application developers [9]. Among these components, one can find numerous different implementations of multi-selection. Different implementations typically exhibit different behavior. These differences exist, not only across platforms, but also across

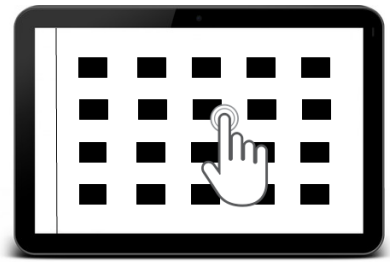


Figure 1.1: Tapping to select on a touch-screen interface.

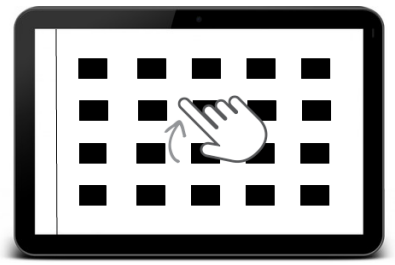


Figure 1.2: Drag/scroll interaction that needs disambiguation.

applications on the same platform. Furthermore, there are inconsistencies and defects in the implementations, which makes it difficult to reason about them or understand their exact behavior. Inconsistent or limited multi-selection and its erroneous implementations can result in poor usability of interfaces, which has been shown to contribute to user frustration [12].

We argue that the issues with multi-selection described above are in large part due to the lack of concrete semantics for it on touch-screen interfaces. This thesis defines such semantics and realizes them as a reusable software library.

Our theoretical foundations are based on the mouse-based multi-selection semantics presented by Järvi and Parent [11]. Adapting and applying their work to touch-screen interfaces presents several challenges. In touch-screen interfaces gestures and taps, as shown in Figure 1.1, serve as the only forms of input; compared to mouse-based selection, there are no modifier keys with which a mode could be defined. It becomes necessary to map combinations of different types of taps (single, multiple, long, and multi-finger taps) and dragging to different selection commands. In addition, one must be careful not to hinder unrelated interactions like scrolling, as shown in Figure 1.2. Further, the design space of gestures is limited by the need to keep user interactions simple and intuitive. The use of complex gestures brings with it problems, such as the lack of discoverability, lack of visibility, accidental activation,

and inconsistency [16].

The remainder of this thesis is laid out as follows. We start with a discussion of related work in multi-selection in Section 2. Section 3 is dedicated to the study of the existing implementations of multi-selection in applications on touch-screen interfaces. Section 4 details some background and important terminology necessary for our touch-screen semantics which we subsequently describe in Section 5. Following that, Section 6 describes the implementation of our multi-selection library. We compare qualitatively, the implementation effort using our framework with that of using alternative frameworks in Section 7. In Section 8, we describe an application we built, describe its use in carrying out a user study, and include several figures to be used in describing the user study in detail. A detailed discussion of the results of the user study follows in Section 9. Finally, this thesis closes by considering future work and presenting our conclusions.

2. RELATED WORK

Multi-selection is a common feature in many applications. Järvi and Parent [11] define precise semantics for mouse-based multi-selection. In a manner similar to Järvi and Parent's work, this thesis focuses on the precise specification and reusable implementation of the commonly established aspects of multi-selection, not on novel inventive ways of selecting elements. In Section 2.1, we explore some of the related work in the general area of multi-selection. Following that, in Section 2.2, we discuss related work pertaining to multi-selection on touch-screen, pen, and gesture-based interfaces.

2.1 Multi-selection

Järvi and Parent [11] introduce and define multi-selection semantics for mouse-based interfaces. They present important terminology in the form of a *selection language* and also a reference implementation in the form of *MultiselectJS*, a JavaScript library. Apart from their work, there is little else on building a cohesive and correct model for selection. Macintosh Human Interface Guidelines [2] introduced extending selections using shift-clicks, and covered some terminology with regard to mouse-based selection, such as defining the terms *anchor* and *active-end* in the context of selections. Similar guidelines for Windows [14] established File Explorer's selection behavior. However, although these guildelines are intended to serve as guidance to the implementer, they are ambiguous at times and offer no help in the implementation of multi-selection.

2.2 Touch-screen Multi-selection

Some prior work has gone into experimenting with different ways to carry out multi-selection on touch-screen devices. Mizobuchi and Yasumura [15] compare circling, a gesture-based method of selection, with tapping in the task of selecting elements with varying levels of cohesiveness and shape complexity. They conclude that circling may be a useful supplement to tapping, with the possibility of developing mixed mode interactions that utilize both. They also note that gesture-based interfaces may allow users to carry out selection tasks more efficiently, particularly when the tasks are more complex than typical selections on two-dimensional grids. Roth and Turner [18] proposed *Bezel Swipe*, a way to perform multi-selections on touch-screens in a manner that does not conflict with other gestures, but at the expense of a slightly steeper learning curve. Leitner and Haller [13] propose *Harpoon Selection* as a novel way to carry out selection tasks on pen-based interfaces. Dehmeshki and Stuerzlinger [6] explore selection from the perspective of perception science and gestalt groups. However, their technique does not appear to be broadly applicable or intuitive enough for the prevalent two-dimensional grids in touch-screen interfaces. Some patents [5, 17] also outline multi-selection operations. These patents describe the causes and effects of different gestures in multi-selection, but not in sufficient detail to constitute a complete specification.

Much of this prior work has gone into developing certain new kinds of multi-selection tools and evaluating their effectiveness. For implementing the conventional and practical multi-selection, all developers have as support are user interface guidelines [3, 8] with their more or less vague definitions of how multi-selection should behave on a particular mobile platform. No prior work defines clear semantics for the general action of multi-selection on touch-screen interfaces, or a specification that

can be used as a blueprint for an implementation free of subtle bugs and inconsistencies.

3. REVIEW OF TOUCH-BASED SELECTION

Touch-screen devices and interfaces are now being used by over a billion people. Platform vendors who write software that runs on these consumer devices define design guidelines which govern interactions like multi-selection, and make available user interface containers that implement these techniques. An example of one such specification in the Material Design Guidelines [8] for both mouse-based and touch-screen user interfaces is shown in Figure 3.1.

Support for multi-selection is strongly recommended for list and grid containers. This does not apply to actions available for a single selection (like a list of phone numbers, where calling is the only action), or if the context requires single-item manipulation (like moving icons on Android's home screen).

Gestures:

- When initiating selection, both long-press and two-finger touch may be extended using a drag gesture to select multiple items. **Items between the beginning and end points of the drag will be included in the selection**.
- On desktop, a drag originating outside the bounds of all items may initiate multi-selection (for example, beginning a drag in the left margin of a list, and extending down and to the right to select list items).

Once an initial selection is made, it can be altered through user actions:

- Touch a selected item to deselect it. Touch an unselected item to select it.
- Shift+touch/click on an item to select all items between two selected points.

Figure 3.1: Material design guidelines by Google.

For example, the bolded text in Figure 3.1 describes a gestural method of performing selection operations on ranges of elements. Upon closer examination, it is found to

be ambiguous. It fails to specify the exact behavior at the start and end points of the drag. It also fails to specify the behavior when there exist elements which have already been selected between the start and end points of the drag. Additionally, there may be user interfaces where items that lie "between" two others items may not be obvious and prone to multiple interpretations. Design guidelines like these are not unambiguous and yet, they are meant to lead to uniform implementations. The study of applications shows that application developers are employing ad-hoc techniques that diverge in behavior. These implementations merit some discussion due to their widespread use in applications. In this section, we discuss some of the common multi-selection models across different touch-screen platforms and applications.

3.1 Selection by Tap

The simplest and the most common multi-selection model observed in applications across different touch-screen platforms is *selection-by-tap*. iOS, Android, Windows 8 and 10, all implement this model in their UI containers. This model of selection starts with the user being placed into a *selection mode*. The user is typically placed in selection mode upon executing a *long-press* on one of the selectables, or upon explicitly selecting an option to turn on a selection mode. The behavior within this selection mode is as follows:

1. Tapping on an unselected element selects it.
2. Tapping on a previously selected element renders it unselected.
3. A *swipe* gesture in any direction scrolls the viewport.

This method is common in both one-dimensional lists of elements and in two-dimensional grids. One example, shown in Fig. 3.3, where this method is used is the "Downloads" application which is packaged with Android OS. When the number of

selections to be made by the user is small (less than ten elements or so), selecting elements one by one may be effective and sufficient. Improvements in precision of touch-screen devices have enabled more complex applications, which may allow for large numbers of selectables and also selections. The selection-by-tap model, however, does not scale well to these larger selections and becomes frustrating and repetitive to use.

Multi-selections typically exhibit the property of spatial locality: elements near a selected element are more likely to be selected also. This property is not taken advantage of in the selection-by-tap model; irrespective of the arrangement or patterns in selections, the model requires tapping each individual element.

(a) Before entering selection mode. (b) Selection-by-tap of individual elements.

Figure 3.2: Downloads application on Android 5.1.

3.2 Google Photos

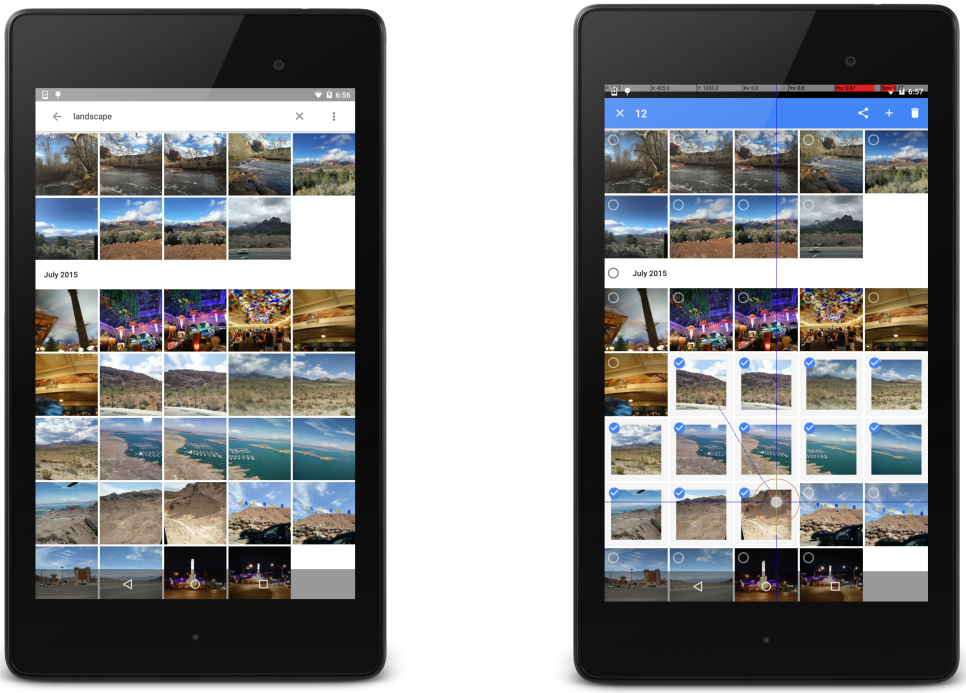
Google Photos [7] is an image organizer and viewer for both Android and Apple's iOS. The application presents images in the form of thumbnails laid out in a two-dimensional grid. A notably improved multi-selection feature was implemented by Google in this application. In a manner similar to *selection-by-tap*, this selection method also starts in a distinct selection mode that is triggered whenever the user long-presses a thumbnail. Once within this selection mode, the behavior is as follows:

1. Tapping on an unselected element selects it.
2. Tapping on a previously selected element renders it unselected.
3. A *swipe* gesture in any direction scrolls the viewport
4. *Long-press* on any element (selected/unselected), followed by a *drag* gesture in any direction selects many elements by rows (Figure 3.3b).
5. A *drag* gesture near the edges of the viewport scrolls the viewport.

This model of multi-selection provides a superset of the features provided by *selection-by-tap*. The drag gesture to select multiple elements is a more convenient method of selection when selecting groups of adjacent elements. It takes advantage of spatial locality of elements to be selected, and is a significant improvement in usability.

Many other applications that offer "advanced" multi-selection capabilities use variations of the method used by *Google Photos*. The applications typically differ in the manner in which the user enters selection mode and sometimes in how they map certain types of dragging within the selection mode to actions other than multi-selection, such as moving or displacing an element. However, these variations are

small and the Google Photos model is used by us as a representative of the state of the art in advanced methods of multi-selection in touch-screen user interfaces.



(a) Before entering selection mode.

(b) Drag selection of many thumbnails.

Figure 3.3: Google Photos version 1.14 on Android.

4. BACKGROUND: MOUSE-SELECTION SEMANTICS

In this section, we discuss briefly selection concepts introduced by Järvi and Parent [11]. The goal of their work is to give precise and unambiguous meanings to multi-selection commands. They observed that multi-selection works differently and often inconsistently in practically all common desktop applications. They argue that this is due to the reimplementation of the same features anew, repeatedly, based on imprecise specifications.

The concepts described in the following sections are used as building blocks in the next section, where we describe the semantics and gesture mappings we designed for touch-screen interfaces.

4.1 Representation

The selectable elements are modeled as an indexed family, $x : I \to M$, with M being the elements, and I being an index set, such as the natural numbers. With this mapping, one can refer to the ith element of a collection, where $i \in I$, and mean the element x_i in the visual representation. The selection state of the indexed set can be represented by $s : I \to \{\mathsf{T}, \mathsf{F}\}$, where $s(i) = \mathsf{T}$ means that x_i is selected and $s(i) = \mathsf{F}$ that it is not. s is called a *selection mapping*. As shorthand for $\{\mathsf{T}, \mathsf{F}\}$, the below sections denote that set as $\mathbf{2}$.

4.2 Primitive Selection Operations

Selection commands such as a command-click and shift-click change the selection mapping, and can be modeled as a function of type $(I \to \mathbf{2}) \to (I \to \mathbf{2})$. Järvi and Parent define *primitive selection operations* as the building blocks from which all other selection operations are built. The primitive selection operation op_J^f applies

the function f to every element in J, and has no effect on elements outside J.

Let $x : I \to M$ a collection, $J \subseteq I$, and $f : \mathbf{2} \to \mathbf{2}$ a mapping. A primitive selection operation is then defined as:

$$\mathsf{op}_J^f : (I \to \mathbf{2}) \to (I \to \mathbf{2}), s \mapsto \lambda i. \begin{cases} f(s(i)), & i \in J \\ s(i), & i \notin J \end{cases}$$

J is the *selection domain*, f is the *selection function* and they together uniquely identify a primitive selection operation. These operations can be composed to realize complex selections. Starting with an empty selection mapping, $e : I \to \mathbf{2}, i \mapsto \mathsf{F}$, if we were to perform a series of selection operations, $\mathsf{op}_{J_1}^{f_1}, \mathsf{op}_{J_2}^{f_2}, \ldots, \mathsf{op}_{J_n}^{f_n}$, the resulting selection mapping can be found by $(\mathsf{op}_{J_n}^{f_n} \circ \mathsf{op}_{J_{n-1}}^{f_{n-1}} \circ \ldots \circ \mathsf{op}_{J_1}^{f_1})(e)$.

There are 4 possible selection functions that can be used in the definition of primitive selection operations.

- $\lambda x.x$, the identity function; $\mathsf{op}_J^{\lambda x.x}$ has no effect on any elements in the selection domain J.
- $\lambda x.\,\mathsf{T}$, that always returns true; $\mathsf{op}_J^{\lambda x.\,\mathsf{T}}$ sets to true all elements in the selection domain J.
- $\lambda x.\,\mathsf{F}$, that always returns false; $\mathsf{op}_J^{\lambda x.\,\mathsf{F}}$ sets to false all elements in the selection domain J.
- $\lambda x.\neg x$, that toggles its argument from true to false and vice versa; $\mathsf{op}_J^{\lambda x.\neg x}$ toggles the selection status of the elements in the selection domain J.

Changes to the selection state of a set of elements can be expressed as a composition of these primitive selection operations. For example, the tapping gestures in Selection-by-Tap (Section 3.1) can be modeled by a $\mathsf{op}_{\{i\}}^{\lambda x.\neg x}$ where i is the index of the element

which was tapped. Similarly, $\mathsf{op}_J^{\lambda x.\,\mathsf{T}}$ can be used to model the selection of several elements with the dragging gesture in Google Photos (Section 3.2). The task of implementing a multi-selection feature can thus be viewed as the task of providing convenient means to specify primitive selection operations.

4.3 Selection Domain and Geometry

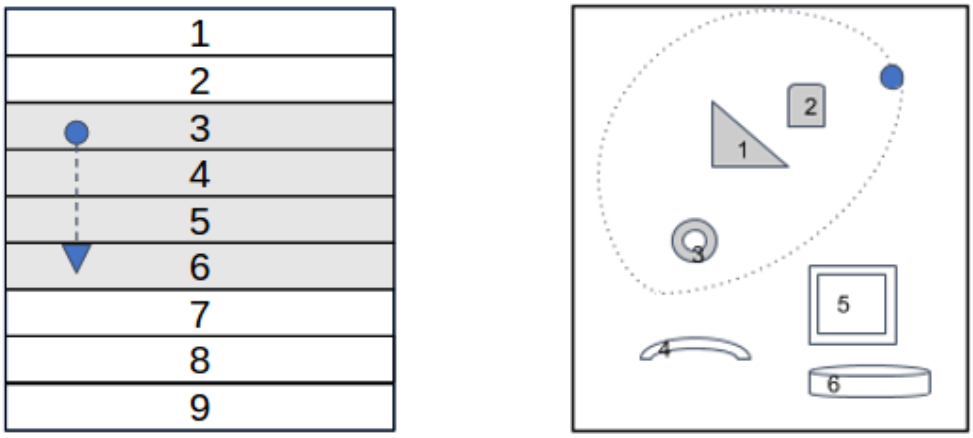
The semantics separate the reusable common aspects of selection from those aspects that vary across applications. The parameter that captures all the varying aspects is the *selection geometry*. The selection geometry comprises two main components, the sdom and m2v functions.

The purpose of the m2v function is to act as a bridge between the different coordinate systems of the pointing device (finger/pen in our case) and that of the selectable elements. The latter coordinate system is called the *selection space*. Sometimes these coordinate systems coincide, but often they do not. For example, in Figure 4.1a, there is a direct translation between the coordinates in the visual space and the index of the element that contains it. In Figure 4.1b, however, some points do not lie on any element. The m2v function allows us to abstract out these differences and transform the sequence of points that the user indicates in the visual space to a sequence of points in the selection space. These points in the selection space are called the *selection path*.

The purpose of the sdom function is to map the selection path into a set of element indices. This set of indices then becomes the selection domain of a primitive selection operation. Often the sdom function only uses the first and last points on the selection path to compute the set of element indices. The first and last elements of the selection path are referred to as the *anchor* and the *active end*. However, there are also systems, such as lasso-style selections, in which sdom may inspect the

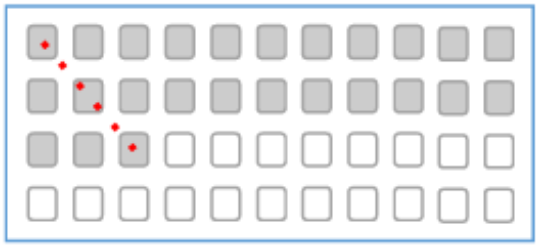
entire selection path to determine the set of indices. The selection domain of the most recent primitive selection operation is called the *active selection domain*.

The effect of the sdom function is exemplified through Figure 4.2. In all three subfigures, points in the selection path are identical, but they lead to the computation of completely different selection domains. In the "row" and "box" selection geometries (Figure 4.2a and Figure 4.2b), the extremities of the selection path coordinates, i.e., the anchor and active end, are used to compute the selection domain. In the "snake" geometry (Figure 4.2c), the entire set of points in the selection path is used to compute the selection domain. In Section 6, we use the same example and detail the differences in the implementation of the sdom function. Similarly to m2v, sdom is provided by the application programmer. These two functions suffice to capture the context and application-specific aspects of multi-selection.

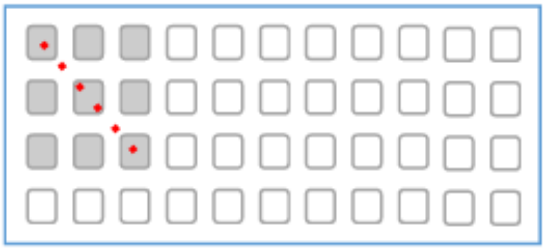


(a) (b)

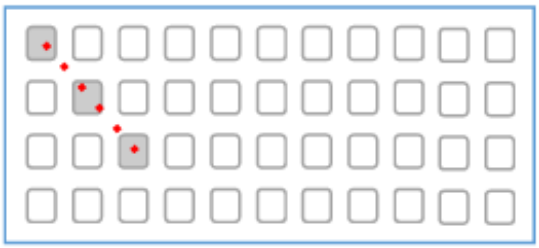
Figure 4.1: The figures show the differences in the coordinate system captured by the m2v function. The blue dot signifies the start of the selection, and the triangle the end of the selection. In (b), the start and the end points coincide. The dotted lines represent points detected during a drag gesture.



(a) The row-wise sdom function.



(b) The rectangular sdom function.



(c) The snake sdom function.

Figure 4.2: These three figures show the effect of the sdom on the selections made. The dotted lines represent points detected during a drag gesture.

5. TOUCHSCREEN SEMANTICS

In this section, we define the semantics for touch-screen interfaces. We first describe a *selection language* in Section 5.1. In the sections which follow, we explain the various parts of our semantics, describe the gesture mappings we designed for touch-screen interfaces and conclude with a discussion of various properties of our semantics.

5.1 Selection Language

5.1.1 Selection State

The *selection state* represents the state of the selectable elements. The selection state consists of three parts:

1. The *base selection mapping* s is a selection mapping as defined in the previous section. This selection mapping is empty in the initial state but may in later states contain baked selections (Section 5.2.2).

2. The *op composition* is a composition of primitive selection operations that have not yet been applied and stored to s. The *op* composition together with the base selection mapping determine the selection state of all selectables. The selection mapping that reflects the currently selected elements is obtained from *op* and s as $op(s)$; the selection state of the i^{th} element can be determined by $ops(s)(i)$. We consider the empty composition, denoted as $op = \cdot$, to be the identity function, so that $\cdot(s) = s$. The selection commands modify the selection state by adding or removing primitive selection operations from the composition, or modifying the few topmost primitive selection operations.

3. The *selection path* P is a sequence of points in the selection space as defined

in Section 4.3. The *selection path* P is what determines the *current selection domain* using the sdom function. Selection commands change the current selection path by adding points to it, setting a new path, or setting it to undefined. The current selection domain is used as the selection domain of each primitive selection operation as it is added to the composition.

Thus, as described above, the selection state can be represented as a tuple of the form $\langle s, op, P\rangle$, where $s : I \to \mathbf{2}$ is the base selection mapping; op the composition of primitive selection operations that have not yet been applied and stored to s; and P the selection path.

5.2 Selection Commands

Tables 5.1 and 5.2 describe the *selection commands* in our selection language. Each *selection command* that we define maps one selection state to a new selection state.

$$
\begin{aligned}
\mathsf{tap}_p &: \langle s, op, \_\rangle &&\mapsto \langle s, \mathsf{op}^{\lambda x.\neg \mathsf{onsel}(p, op(s))}_{\mathsf{sdom}(\cdot|p)} \circ \mathsf{op}^{\lambda x.x}_{\emptyset} \circ op, \cdot|p\rangle \\
\mathsf{double\text{-}tap}_p &: \langle s, \mathsf{op}^{f}_{\_} \circ op, P\rangle &&\mapsto \langle s, \mathsf{op}^{f}_{\mathsf{sdom}(P|p)} \circ op, P|p\rangle \\
\mathsf{double\text{-}tap}_p &: \langle s, op, \_\rangle &&\mapsto \langle s, \mathsf{op}^{\lambda x.\,\mathsf{T}}_{\mathsf{sdom}(\cdot|p)} \circ \mathsf{op}^{\lambda x.x}_{\emptyset} \circ op, \cdot|p\rangle
\end{aligned}
$$

Table 5.1: Basic selection commands.

$\mathsf{undo} : \langle s, o_n \circ o_{n-1} \circ op, \_ \rangle$	$\mapsto \langle s, op, \bot \rangle$
$\mathsf{undo} : t$	$\mapsto t$
$\mathsf{bake} : \langle s, op \circ o_4 \circ o_3 \circ o_2 \circ o_1, P \rangle$	$\mapsto \langle \mathsf{store}(s, o_2 \circ o_1), op \circ o_4 \circ o_3, P \rangle$
$\mathsf{bake} : t$	$\mapsto t$

Table 5.2: Undo and Bake.

We briefly describe next the interpretation of these selection commands. While function composition has its regular meaning, we also assume that the composition structure within *op* is accessible, so that pattern matching can extract individual primitive selection operations. When we discuss the meaning of the selection commands in our selection language, we use the metavariable o to range over primitive selection operations and *op* over compositions of zero or more primitive selection operations. For example, the pattern $o_1 \circ o_2$ matches compositions that have exactly two primitive selection operations, binding o_1 to the first and o_2 to the second. The metavariable *op* can appear on either end of a pattern: $op \circ o$ and $o \circ op$ both match compositions that have one or more primitive selection operations. The symbol $\cdot$ denotes the empty sequence. The operator | is a context-dependent function whose meaning can vary, but to simplify, we assume for now that it extends a sequence with a new point. The metavariable _ binds to anything and signifies an unused value. Finally, we use the symbol $\bot$ to signify an undefined value.

All selection commands of the selection language maintain the invariant that the composition of primitive selection operations has an even number of elements. This is a technicality that makes the definition of the language more concise at the expense

of sometimes adding extra empty operations to the composition. A possible valid initial empty selection state that satisfies the invariant is $\langle e, \cdot, \bot \rangle$, where e is the empty selection mapping.

5.2.1 Basic selection commands

The two basic selection commands tap and double-tap are presented in Table 5.1. The definitions for tap and double-tap are very similar to the definitions of c-click and s-click, respectively, in Järvi and Parent's mouse-based multi-selection semantics. The functions double-tap, and tap are named so that their meanings are close to the gestures we define later in this section. The point parameter p to each operation is written as a subscript, as in tap_p, to keep it notationally separate from the selection state parameter.

The effect of a tap is to toggle the selection state of an element. Further, it resets the current selection path and sets the tapped selection space point to become the new anchor. The tap command accomplishes this by adding a pair of primitive selection operations to the composition in op. It uses $\lambda x.\,\mathsf{T}$ or $\lambda x.\,\mathsf{F}$, depending on whether p is selected or unselected. $\mathsf{onsel}(p, s) = \textbf{if}\ \mathsf{sdom}(\cdot|p) = \{i\}\ \textbf{then}\ s(i)\ \textbf{else}\ \mathsf{F}$ is a helper function that determines whether p is on a selected or unselected element in the selection mapping $op(s)$. Although one primitive selection operation would suffice to achieve the effect desired from a tap, a second identity operation is added to maintain the invariant of even number of primitive selection operations in the op composition.

The effect of a double-tap is to select or deselect a range of elements. It adds points to the selection path using $|$. Whether a double-tap performs selection or deselection depends upon last tap that was performed. If the last tap selected an element q, the following double-tap commands continue to perform selections and

adding to the selection path using |. Similarly, if q was deselected by the last tap, any following double-tap commands perform deselections. If there isn't an anchor established already, the effect of a double-tap is the same as that of a tap, except that the mode is always set to "selecting".

Concretely, double-tap has the above mentioned behaviors shown in the semantics as two separate cases. The first matches if both the *op* composition is not empty and the selection path is defined. A new selection domain is computed from this path to replace the selection domain of the outermost primitive selection operation in the composition. In the second case, there may be no selection path to extend or no outermost primitive selection operation, so a new path is established, and a new pair of primitive selection operations added.

5.2.2 Bake and undo

The bake and undo selection commands were introduced by Järvi and Parent in the mouse-selection semantics and we retain them. In this section, we describe briefly the working of these two commands.

One benefit of representing selections as compositions of primitive selection operations is that it becomes easy to define and implement an undo operation for selections. Practically no applications today provide such a feature, but we believe it can be very useful, though more so with mouse-based multi-selection. It is not an uncommon scenario that one constructs a complex selection of, say, photo thumbnails, and loses the entire selection with a single mis-click. An undo would be very handy in such a situation. In touch-based selection, we do not provide a gesture that could wipe out an entire selection, but a careless drag can nevertheless bring a user to an undesired selection state.

The effect of undo is to roll back *op* to that in the previous selection state prior

to the execution of the last basic selection command. While undo restores the active domain of the previous selection state, it sets the selection path to the undefined value $\perp$. This is because the selection domain of the restored primitive selection operation was not computed from the current selection path, and thus keeping the current path would make a subsequent double-tap command unpredictable. An undefined path forces double-tap to add a new pair of primitive selection operations, avoiding the surprising behavior. The second case of undo is an identity function; it is applied if there are no operations to undo.

So that the memory requirement of the undo feature can be constrained, Järvi and Parent define the bake selection command and we retain it with nearly no modification. It is used for reducing the size of the composition, without changing which elements are selected. It extracts the two least recently added primitive selection operations from *op* and "bakes" their effect permanently to the base selection mapping s; store(op, s) is an operation that constructs an efficient selection mapping from *op* and s. The second case of bake is an identity function; it is applied if fewer than two primitive selection operations would remain in *op* after baking. Emptying the composition completely would bake the operation that defines the active domain and thus change double-tap's behavior.

5.3 Gestures

We observed that the gestures could be made less complex by introducing a *selection mode*, akin to *Selection-by-Tap* and *Google Photos*. It varies from application to application how the user enters selection mode. Typically, it is entered by a long-press on the selectable elements or using an explicit toggle on the user interface. Once the user enters such a mode, all interactions are interpreted as selection commands.

The formal semantics and gestures go hand in hand. The most basic gestures as

defined by our semantics are tap and double-tap. We first describe the effects that they have and then describe some additional gestures we defined.

The tap gesture serves both to invert the selection state of a selectable, and to set the current *mode*. The mode parameter is defined as being one of "selecting", or "deselecting", depending on the action performed by the last tap operation. In the "selecting" mode, all double-tap operations select elements, and in the "deselecting" mode, they deselect elements. The mode parameter starts as "selecting" when the system is initialized. The *anchor* is set only by tap operations. Any subsequent double-taps maintain the same *anchor* and only add or modify other elements in the selection path.

We define two additional gestures which we use to perform selection and deselection on ranges of elements. These gestures are as follows:

- A double-tap followed by a drag operation. Each point of the drag is considered a new double-tap operation. This allows for a drag gesture to extend a selection or deselection without the need for any additional selection commands.

- A long-press immediately followed by a drag operation. The long-press on a selectable is interpreted as a tap, which sets an anchor and changes the mode as defined previously. The drag gesture that follows is interpreted as a series of double-taps.

We carried out experiments with several alternative gesture mappings. One such mapping used gestures involving multiple simultaneous touches on the screen. These gestures, also called *multi-touch gestures*, expand the design space of selection gestures. As multi-touch gestures are not typically used by applications, we could even eliminate the need for a specialized selection mode. However in practice, multi-touch gestures are difficult to perform precisely when the size of elements approaches the

size of the finger or pointing device. Touch-screen devices come in various sizes and have different levels of sensitivity. It would prove difficult to write generic methods which work well on all of them. The multi-touch gestures also proved unintuitive, and jarringly different from the existing common selection mechanisms. Due to the reasons we presented above, we chose to avoid the use of multi-touch gestures or other novel selection techniques.

The gestures we described above which derive from tap and double-tap realize the commonly established aspects of touch-screen multi-selection. We believe that they offer the right balance between simplicity, familiarity and efficiency in approaching the task of multi-selection.

5.4 Properties of our Selection Semantics

When implementing multi-selection feature based on our semantics, several beneficial properties come "for free". In this section, we outline these properties.

5.4.1 Selection state preserving active domain

In order to describe this property, we first contrast it against the alternative "advanced" multi-selection feature offered by Google Photos. We also remind the reader that the *active selection domain* or *active domain* is the selection domain of the most recent primitive selection operation, computed from the selection path.

Under Google Photos, drag gestures to select multiple elements may affect selections made by a previous selection operation and may lead to their (possibly unintended) deselection. This is made clearer in Figure 5.1. The sub-figures show a progression of events that leads to some previously selected elements being deselected by a modification of the *active domain*.

Our semantics adopt a more consistent approach in which elements are deselected only by an explicit gesture executed by the user to do so, and not implicitly

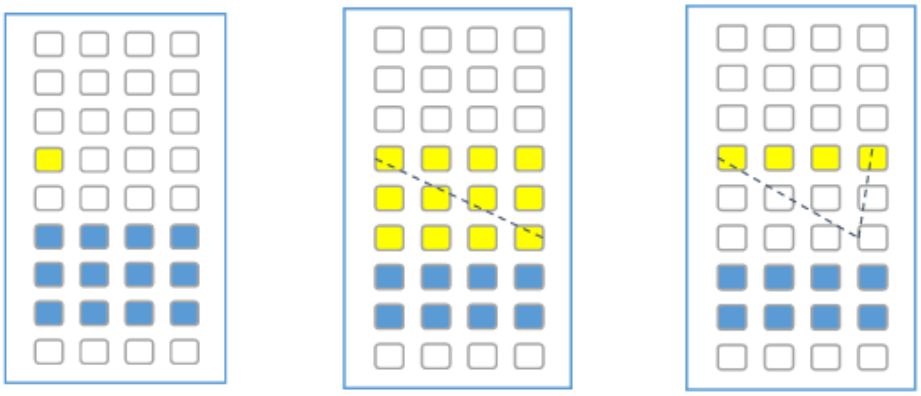


Figure 5.1: These three figures show a progression of events explaining the "selection state erasing active domain" of the Google Photos selection mechanism. In (a), the elements marked in blue have already been selected and a new drag is started from the element marked in yellow. In (b), the dotted line shows the drag gesture performed. In the state shown, the drag has not been released by the user. Finally, (c) shows that the fourth row from the bottom has been deselected by the change in the active domain indicated by the dotted line.

by changes in the active domain. Thus, this may prevent accidental deselections. Figure 5.2 shows a progression of events similar to those in Figure 5.1, but now with our semantics. It can be clearly seen that in contrast with Google Photos, our semantics do not allow the active domain to affect selections made by a previous set of selection operations.

Due to this key difference in the way our semantics behave when the active domain intersects with a previous selection, we refer to our touch-screen semantics as having a *selection state preserving active domain*. We refer to Google Photos and other similar selection mechanisms as having a *selection state erasing active domain*.

5.4.2 Equivalence of double-tap and drag operations

One common scenario is multi-selection on a large range of elements spanning several screens. Typical multi-selection frameworks featuring gestures for selecting ranges support this by allowing the user to start a drag gesture and "push" it against

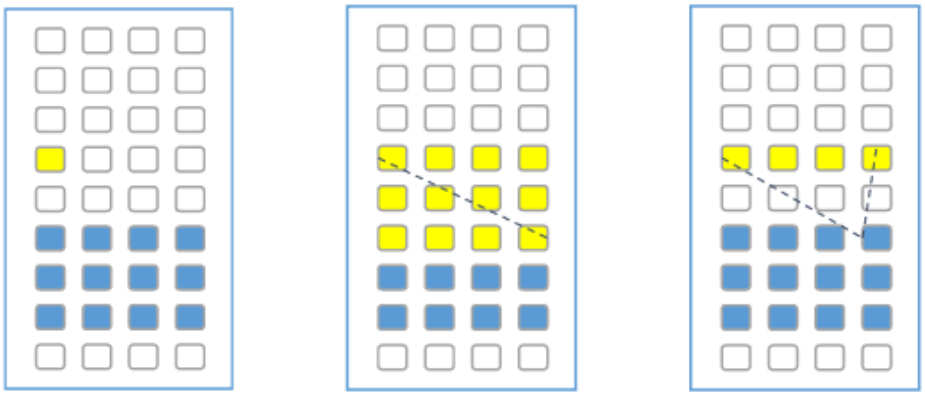


Figure 5.2: These three figures show a progression of events explaining the "selection state preserving active domain" of our semantics. In (a), the elements marked in blue have already been selected and a new drag is started from the element marked in yellow. In (b), the dotted line shows the drag gesture performed. In the state shown, the drag has not been released by the user. Finally, (c) shows that the previous selection is completely preserved after the completion of the drag.

an edge of the viewport, which causes the viewport to scroll and extend the drag gesture beyond the bounds of a single screen of elements. We argue that this is inefficient and often leads to unintended effects.

With our semantics, it is possible to split a single range-selection operation over several gestures. One way to perform the same task with our semantics could be to mark an anchor at the beginning of the selection by executing a tap, scrolling by any means desired and then using a double-tap to set the active-end and select that range. The range can then be modified again by either another double-tap, or a double-tap followed by a drag. The equivalence we define between a double-tap and dragging serves thus not only to simplify the semantics, but also leads to a convenient method of carrying out complex selections.

5.4.3 Deselection of ranges of elements

As described in Section 5.3, the semantics offer the ability to perform deselections over entire ranges of elements. This property, while being a natural counterpart to performing selections over ranges of elements, is rare in contemporary multi-selection implementations.

5.4.4 Undo and redo

As previously mentioned, in the absence of a command akin to "click" in mouse multi-selection, the cost of performing an incorrect selection gesture on a traditional rectanglular grid or list of elements is likely to be considerably smaller in touch-screen interfaces. Our semantics, however make undo trivial to implement. Hence, the implementer may choose whether to provide undo and redo functionality their application.

The user may, for example, perform deselections where selections are intended, and for such cases, undo and redo may prove useful to be able to roll back to a previous selection state. There may also be certain complex arrangements of elements, such as in a vector drawing or image editing applications, where the provision of an undo command could make multi-selections easier.

6. IMPLEMENTATION

We provide an implementation of our touch-screen semantics for the Android platform. We note that Android was chosen simply due to our familiarity with Java and the Android SDK, and that it would have been approximately the same effort to develop a library implementation for a different platform such as iOS or Windows. A class diagram shown in Figure 6.1 shows the most important classes and interfaces that make up our implementation. Structurally, our library bears resemblance to the reference implementation, *MultiSelectJS* [10], provided by Järvi and Parent.

The sections below outline the important parts of our implementation. We also describe clearly the parts that the application programmer must provide versus the services that the library provides. We explain the behavior of selection semantics in different selection contexts with code samples and examples.

6.1 Selection State

The `SelectionState` class is the central component of the library. It encapsulates all the functionality provided by our semantics. The different members of the *selection state tuple* described in the formalism are private members of this class. These members are:

- A `SelectionMap` object to store the base selection mapping.
- An `OpComposition` object to store the *op* composition.
- A `Path` object to store the selection path.

The primitive selection operations are maintained in the `OpComposition` object that contains an array of `Op` objects. The size of the selection state object is

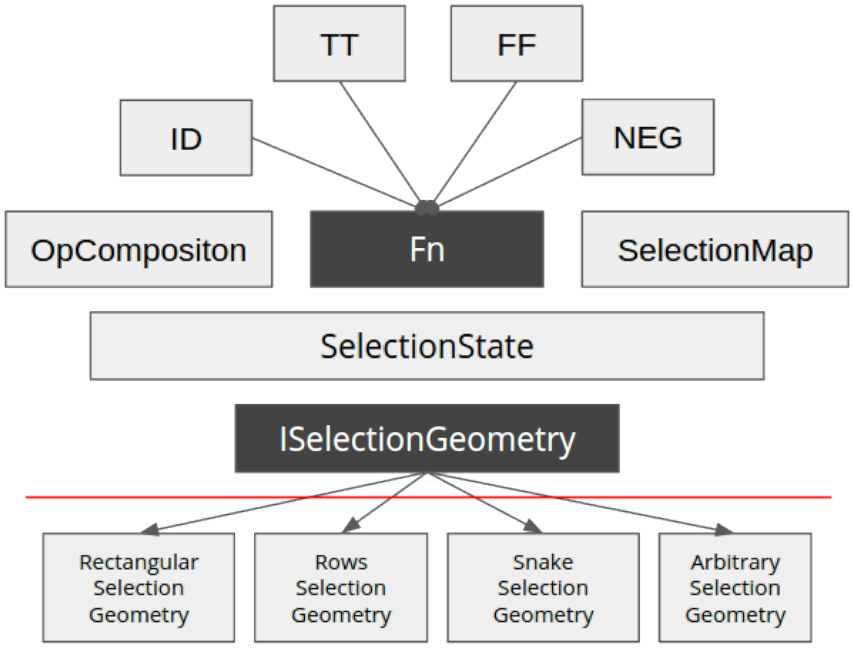


Figure 6.1: A diagram featuring the important parts of Multiselect-Android. The parts above the red line are provided by the library and the parts below it are typically written by the implementer. The darker boxes denote interfaces and the lighter ones denote classes. An arrow between a class and an interface implies that the class implements that particular interface.

proportional to the total number of elements in all of the primitive selection operations' domains in the composition. The `SelectionMap` class is used to store the selection domains as well as selection mappings, and is at its core a Java `HashSet`. The indices present within the set are considered to be in the selected state T, and those not in it are in the unselected state F. This representation in the form of a `HashSet` makes lookup, updation and deletion straightforward.

The `SelectionState` class provides as public methods all selection commands of the selection language that we defined in Table 5.1 and Table 5.2 except the `bake` method, which is private and called internally by the library when required. These

methods modify the internal selection state according to rules defined in the selection language. The `SelectionState` class constructor is as shown below.

```
SelectionState(ISelectionGeometry geometry, IRefresh rc, int
maxUndo)
```

The `ISelectionGeometry` interface is implemented by all selection geometry classes. The `IRefresh` interface consists of a single method, `refresh`. This callback is specified by the user and it updates the state of the GUI in response to changes in the selection state of the elements computed by the library. The `maxUndo` argument is an integer which specifies the maximum number of undo states that should be maintained. It limits the size of the `OpComposition` and decides when the `bake` method must be called to compact the `OpComposition`.

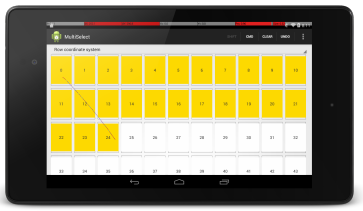
6.2 Selection Geometry

Selection state objects are parameterized by a selection geometry object that contains the application-specific aspects of multi-selection. The geometries are defined according to an interface named `ISelectionGeometry`. The functions that it defines are:

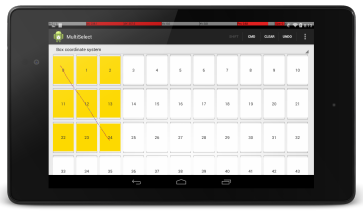
- `m2v(Point p)` to translate screen coordinates into selection space coordinates.
- `extendPath(Path spath, Object vpoint)` to add a selection space point to the selection path.
- `selectionDomain(Path spath)` to compute a set of indices from the se-

lection path.

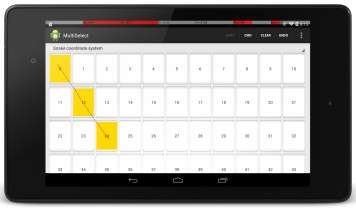
The selection geometry implementation remains faithful to the function definitions made in Section 4.3. The `m2v` method corresponds to the m2v function and the `selectionDomain` method to the sdom function. The `extendPath` method is an implementation of the | operator defined in Section 5.2.



(a) The row selection geometry.



(b) The box selection geometry.



(c) The snake selection geometry.

Figure 6.2: These three figures show the effect of the selection geometry on the selections made. All three figures show an identical *drag* gesture between elements 0 and 24. Special logging was turned on at the system-level to visualize the touch-gestures on screen.

Snapshots of an implementation of the *row-wise*, *rectangular* and *snake* geometries in an Android application using our library are shown in Figure 6.2. It is important to note that there is very little change in the code from the perspective of the implementer in realizing these three geometries. The definitions of these functions for the row-wise selection geometry are shown in Listing 6.1.

Listing 6.1: `m2v`, `sdom`, `extendPath` for the row-wise selection geometry.

```
@Override
// returns a nullable index of the element that was hit.
public Object m2v(Point p) {
  for(int i = 0; i < parent.getChildCount(); i++) {
    View child = parent.getChildAt(i);
    Rect bounds = new Rect();
    child.getHitRect(bounds);
    if (bounds.contains((int)p.X, (int)p.Y)){
      child.invalidate();
      return child.getId();
    }
  }
  return null;
}

@Override
public SelectionMap selectionDomain(Path spath) {
  SelectionMap sm = new SelectionMap();
  // If there isn't anything in our path, just return the empty
     map.
  if(spath.size() == 0) return sm;
  int b = Math.max(0, Math.min(anchor(spath), activeEnd(spath)));
  int e = Math.min(this.size - 1, Math.max(anchor(spath),
     activeEnd(spath)));
  for (int i = b; i <= e; ++i) sm.set(i, true);
  return sm;
}

@Override
public boolean extendPath(Path spath, Object point) {
  if(point == null) return false;
  if(spath.size() == 2){ spath.pop(); }
  spath.push(point);
  return true;
}
```

Some of the salient features of the code shown in Listing 6.1 are as follows:

- The `m2v` method first iterates over the *children*, i.e., the selectable elements within the parent container. Our library requires that the selectable elements define a public method `getId()` within them, which is called to map each selectable to an index. For each selectable element, the `m2v` method checks if the point of interaction fell within that element's bounding box. In a system in which each point in the visual space maps deterministically to a selectable

element, the `m2v` function could be written more efficiently as a direct mathematical transformation into an index, requiring no iteration over elements.

- The `extendPath` method adds elements to the selection path. The row-wise geometry simply requires the first and last points of the selection path, i.e. the *anchor* and the *active end* to compute the selection domain. Hence we perform an optimization such that the size of the selection path is constrained to no more than two elements at any given time.

- The `selectionDomain` function computes the selection domain from the selection path. `anchor` and `activeEnd` are helper functions which extract the first and last elements of the selection path. The rest of the computation is creating an empty `SelectionMap` and adding to it the various indices that form the selection domain.

If we were to contrast this with the implementation of the "snake" selection geometry, we see that the `m2v` function remains the same. The new `selectionDomain` and `extendPath` functions are shown in Listing 6.2. As can be seen, they are not very different from the functions we defined for the row-wise selection geometry. These minor changes are sufficient to realize completely different behavior shown in Figure 6.2c. The presented code snippets are representative of the typical complexity of these functions when implementing common multi-selection behaviors.

Listing 6.2: `sdom`, `extendPath` for the snake selection geometry.

```
@Override
public SelectionMap selectionDomain(Path spath) {
  SelectionMap sm = new SelectionMap();
  // If there isn't anything in our path, just return the empty
      map.
  if(spath.size() == 0) return sm;
  for(int i : spath.getPath()){
    sm.set(i, true);
  }
  return sm;
}

@Override
public boolean extendPath(Path spath, Object point) {
  if(point == null) return false;
  spath.push(point);
  return true;
}
```

6.3 Gestures and Commands

Public methods `tap(Point p)` and `double_tap(Point p)` are provided by the `SelectionState` object, and correspond to the basic selection commands in our semantics. The Point `p` is a point in the selection space obtained from executing `m2v` on the coordinates in the visual space. The selection commands must be invoked as results of the corresponding gestures on the GUI in selection mode. This may be achieved by using the `GestureDetector` class provided by the platform. We implement such a `GestureDetector` within the parent container in order to listen for all events of interest to us, and then dispatch method calls to selection commands with their respective coordinates. The refreshing of the GUI after the execution of each selection command is performed by the library using the `refresh` callback passed in to the `SelectionState` object during its construction.

In this section, we described in detail our implementation of our semantics in the form of a library. Next, we compare our library against other competing frameworks from the perspective of the implementer of the multi-selection feature.

7. COMPARISON OF IMPLEMENTATION EFFORT

In this section, we first, in Section 7.1, describe the general process of developing a multi-selection feature with existing tools provided by platform and third-party library vendors. We then, in Section 7.2, compare it with the effort required for a developer to implement a multi-selection feature using our library.

7.1 Vendor and Third-party Implementations

The commonly used method of implementing a multi-selection feature in an application is to make use of the built-in containers offered as part of the SDK by platform vendors. The two most prominent platforms are Google's Android and Apple's iOS.

The Android SDK provides a built-in `GridView` container that allows users to position elements in a two dimensional grid. The `GridView` container has a property that can be set to allow multiple elements to be selected and in this mode the container tracks the selection state of each individual element. However, it only provides the Selection-by-Tap mechanism of selection and provides no gestures to select more than one element using a single gesture. Although efficient multi-selection is becoming more common and necessary, this built-in container requires the developers to re-implement such functionality in their applications. A more generic `ViewGroup` class is also provided, which allows unrestrained positioning of elements instead of a two dimensional grid. The `ViewGroup` class provides even less functionality and does not provide any mechanism to deal with selections. It requires that the application developer implements anew even a simple Selection-by-Tap selection model.

In iOS, a study reveals a similar scenario with the `UICollectionView` con-

tainer. It features a similar API and any gesture based multi-selection requires additional effort and custom logic using the `GestureRecognizer` classes.

There are some third party libraries, such as *drag-select-recyclerview* [1], that provide some advanced gesture-based multi-selection for Android. We have found that libraries of this nature are inflexible and typically severely limited in the types of geometries they can handle. They also use complex logic that makes debugging and testing them difficult.

A common and universally required feature like multi-selection is made more complex by the lack of convenient and easily testable ways to implement it.

7.2 Multiselect-Android

Multiselect-Android that we presented in the previous section is a significant improvement over the existing multi-selection frameworks for touch-screen interfaces. Our approach abstracts out complex but reusable parts of multi-selection. This allows the implementer to focus on three methods: `m2v`, `extendPath`, and `selectionDomain`. These methods are stateless and easy to test independently.

In our study of applications, we found that although there is some variation in the nature of multi-selection features, a vast majority of them are list or grid-based. We thus provide `MultiSelectGridView` and `MultiSelectListView` containers that we designed as convenient user interface containers for implementers. They are derived from the user interface containers provided by the platform vendor, `GridView` and `ListView`, and can be used as drop-in replacements. We also provide a more generic `MultiSelectLayout` that is generic and allows any arrangement of child views that do not necessarily conform to a list or a grid. The use of `MultiSelectGridView`, `MultiSelectListView`, or `MultiSelectLayout` does not require the implementer to write gesture detectors or callbacks to various

selection commands.

The flexibility of our approach to multi-selection is showcased by the ease of implementing geometries like that shown in Figure 7.1. Although it appears considerably different from the grid-based examples, it uses the same underlying semantics and requires relatively simple selection geometry functions.

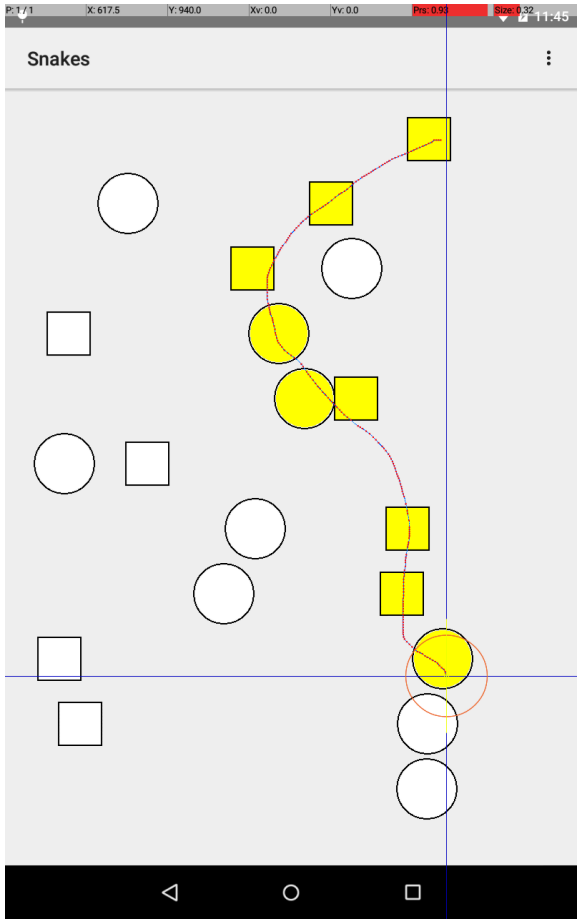


Figure 7.1: A screencapture showing an application implemented using Multiselect-Android. The application allows multi-selection of custom shapes arranged in an arbitrary fashion.

Based on the above observations, we find that Multiselect-Android allows for the

reliable and sound implementation of multi-selection on a wide variety of element arrangements. Vendor-provided and third-party multi-selection features are typically lacking, inflexible, and difficult to test. Multiselection-Android also requires relatively less application-specific logic which is completely encapsulated in the selection geometry.

8. USER STUDY

In this section, we describe a user study we carried out to measure the effectiveness of our selection semantics and library. The user study was carried out using an application we designed for Android. This application made use of the *Multiselect-Android* library. We start with a general description of our study, followed by descriptions of the individual selection tasks we designed for users to perform in Section 8.4. Finally, Section 8.5 concludes by detailing the data we collected during our study.

8.1 Goals

The goals of our user study were:

1. Evaluate the effectiveness of *Multiselect-Android* in carrying out selection tasks of varying complexity.

2. Compare our touch-screen semantics with other common approaches: *Selection-by-Tap* and *Google Photos*, that we discuss in Section 3.

8.2 Implementation

In order to carry out the user study, we designed an application that included implementations of all three selection techniques, *Selection-by-Tap*, *Google Photos* and *Multiselect-Android*. Screen captures from this application are shown in Appendix A.

Our library implementation, *Multiselect-Android*, realizes by default the selection semantics that we described in earlier sections. However, in order to test *Selection-by-Tap* and *Google Photos*, we created modified versions of the library with slightly different selection semantics, which we detail below.

8.2.1 Selection by Tap

Selection-by-Tap was relatively easy to implement because its functionality is a subset of that provided by Multiselect-Android. When we consider the semantics, the only rule that we needed to implement for *Selection-by-Tap* was tap. The definition is identical to that in Table 5.1. A modification to the library that disables the double-tap command was sufficient for realizing this selection mechanism.

8.2.2 Google Photos

The modification needed for *Google Photos* style of multi-selection is considerably more complex. It retains the tap command but has no notion of a double-tap. It does, however, allow performing drags. To account for this variation, we define a new selection command drag, shown in Table 8.2.2.

$$
\begin{aligned}
\mathsf{tap}_p &: \langle s, op, \_ \rangle && \mapsto \langle s, \mathsf{op}^{\lambda x.\neg \mathsf{onsel}(p, op(s))}_{\mathsf{sdom}(\cdot|p)} \circ \mathsf{op}^{\lambda x.x}_{\emptyset} \circ op, \cdot|p \rangle \\
\mathsf{drag}_p &: \langle s, \mathsf{op}^{f}_{j} \circ op, P \rangle && \mapsto \langle s, \mathsf{op}^{\lambda x.T}_{\mathsf{sdom}(P|p)} \circ \mathsf{op}^{\lambda x.F}_{j} \circ \mathsf{op}^{f}_{j} \circ op, P|p \rangle \\
\mathsf{long\text{-}press}_p &: \langle s, op, \_ \rangle && \mapsto \langle s, \mathsf{op}^{\lambda x.\,\mathsf{T}}_{\mathsf{sdom}(\cdot|p)} \circ \mathsf{op}^{\lambda x.x}_{\emptyset} \circ op, \cdot|p \rangle
\end{aligned}
$$

Table 8.1: Selection commands for Google Photos.

The tap function is again identical to that in *Multiselect-Android*.

The long-press function serves to initiate a range selection. Upon close observation of the selection commands, it is seen that the long-press command is identical to the tap gesture except that it always selects the element it is invoked upon. In contrast, tap may select/deselect an element depending upon the selection state of the element it is invoked upon.

The drag command is invoked when a *swipe* gesture occurs immediately following the execution of a long-press command. At every point in this swipe gesture, the drag command is invoked with the corresponding selection space point p. The drag command as defined in the semantics adds two new primitive selection operations into the *op* composition upon each invocation. The first of these operations deselects the entire domain from a previous drag, and the second sets up a new selection. It is this combination of primtive selection operations that allows changes in the active domain during a drag operation to modify selections which were completed prior to the start of the drag operation. This particular set of operations realizes the *selection state erasing active domain* of Google Photos described earlier in Section 5.4.1.

The above defined semantics work to realize the Google Photos behavior of selection. Making use of these commands, we created a modified version of our library to realize this selection behavior.

8.3 Study Setup

The application we designed for the user study presented the user with one practice screen and four selection tasks, all to be performed using the same selection mechanism, one of the three discussed in the previous section. The assignment of the selection mechanism to users was carried out in a random fashion. Each user had to complete all four tasks and answer a short questionnaire about their experience at the end. The application was presented to users on a Nexus 7 (2013) tablet running Android 5.1.1. We recorded detailed data of the user interaction during each task. From this data, we derived the selection time and accuracy metrics, which are of primary interest to us. The data we recorded is described in detail in Section 8.5.

8.4 Tasks

Regardless of which selection mechanism was assigned to a user, the application behaved exactly the same in all other respects except the selection mechanism. The users were first presented with concise printed instructions on how the selection mechanism that was assigned to them works. The user was instructed that at the end of each task, they could tap a "Done" button to proceed to the next task. The user thus explicitly signalled that the task was completed. The instructions for each task were presented within the application before the beginning of each task.

In all figures below, depicting the selection tasks, the color scheme is as follows:

- Blue outline is the viewport at the beginning of the task. In order to access elements outside the viewport, the user had to scroll using a *swipe* gesture.
- Gray boxes denote selectable items.
- Blue boxes denote selectable items that the user was instructed to select in that task.
- Orange boxes denote items which were pre-selected at the beginning of the task and had to stay selected at the conclusion of the task.

After the users were given the printed instructions, they were handed the device running the user study application. They were first requested to input their age and gender, which were both optional fields. Following that, they were presented with what we call "Task 0", a collection of file icons on a grid. The layout of the task is shown in Figure 8.1a. In this task the users could familarize themselves with their assigned selection mechanism; we encouraged them to try out each selection command before moving on to the actual selection tasks. We collected data from

this practice task as well, mainly to know the time taken by the users for learning the selection mechanism.

8.4.1 Task 1

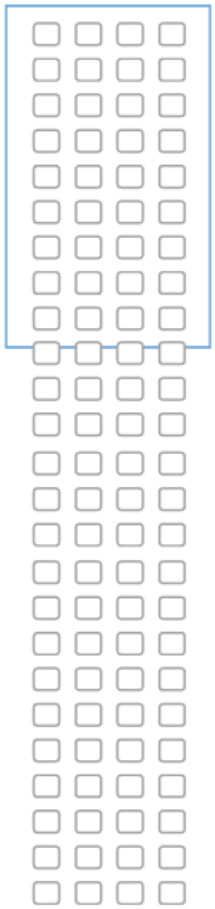
Task 1 was intended to compare the selection mechanisms when presented with the simplest of selection tasks; selecting three distinct elements that have no spatial ordering. The layout of this task is described in Figure 8.1b. The items in this task fit completely within the viewport and hence the task did not require scrolling.

8.4.2 Task 2

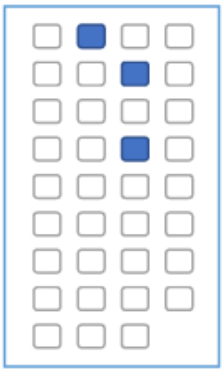
Task 2 spanned several pages and hence required scrolling. The layout is as shown in Figure 8.2a. In the instructions screen preceding the start of this task, we reminded the user that there were more than one screens of items. Both *Google Photos* and *Multiselect-Android* provide gestures to select elements by rows. We intended to measure the effectiveness of such gestures. This selection pattern is commonly seen in gallery and file-explorer applications that sort items based on similarity in type, name, time, and so on, which may lead to good spatial locality.

8.4.3 Task 3

The layout of Task 3 is as shown in Figure 8.2b. In the instructions screen presented prior to the start of the task, the user was informed that there were some elements which had been selected prior to the start of the task. This task was intended to investigate the effects of the choice between the *selection state preserving* and *selection state erasing* active domain on the performance of selection tasks. The concepts of *selection state preserving* and *selection state erasing* active domain are described in Section 5.4.



(a) Task 0.

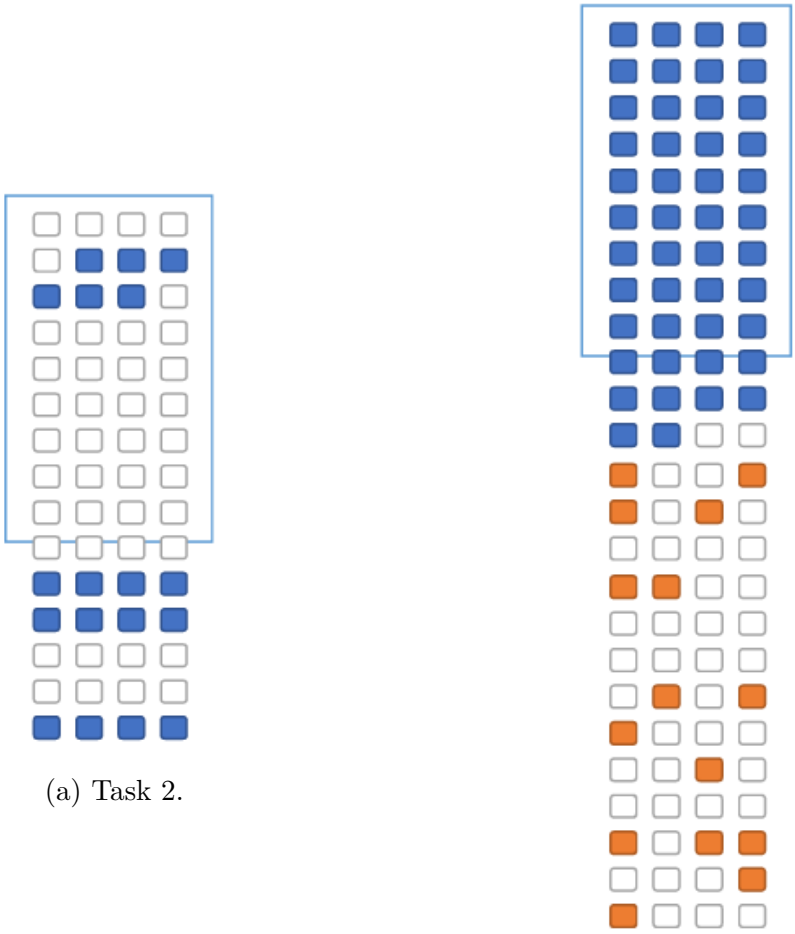


(b) Task1.

Figure 8.1: Tasks 0 and 1.

8.4.4 Task 4

Task 4 presented the user with a different selection geometry, one that allows for selections in a rectangular, rather than row-wise fashion. The change in geometry was explained to the user in the instruction screen prior to the start of the task. The layout of elements for this task was as shown in Figure 8.3. This task was designed for studying the use of gestures for deselection. The rectangular selection geometry



(a) Task 2.

(b) Task3.

Figure 8.2: Tasks 2 and 3.

allows us to measure this particular aspect better than the row-wise selection geometry used in previous tasks.

8.4.5 User feedback

At the conclusion of the four tasks, the users were presented with two simple feedback questions. The questions were:

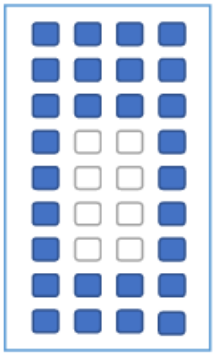


Figure 8.3: Task 4.

1. Was the method sufficient for performing the required tasks?

2. Was the method frustrating to use?

The questions required a boolean yes/no answer and were chosen to deal with two separate aspects of the user's opinion on the selection mechanism. The first deals with the mechanism's adequacy for performing the tasks given. The second deals with the user's reaction to performing tasks using it.

8.5 Metrics

We recorded the following pieces of data to help us compare the three selection mechanisms.

1. Age and gender of the user.

2. The selection mechanism that was assigned to the user.

3. A log of "events", such as taps and drags, with corresponding X-Y coordinates, timestamps, and element indices.

4. The final selection state at the time the user tapped “Done” on each task.

5. The users’ answers to the feedback questions.

With the help of this raw data accrued, we carried out the statistical analyses which are detailed in the following section.

9. RESULTS

This section describes our hypotheses and presents the statistical analysis of the data we recorded from the study. We describe the demographic in Section 9.1. Section 9.2 describes the transformations of raw data we performed to extract useful data. Section 9.3 details the statistical analyses we performed on that data.

9.1 Demographic

Our user study involved 36 users of ages between 18 and 38. 24 of them identified as male and 11 as female in the study. One participant did not wish to disclose their age or gender. The demographic was already familiar with the use of touch-screen interfaces and with the notion of multi-selection. The users were randomly split into three groups and each group performed selection tasks with one of the three mechanisms being compared. In the following text, we use *selection technique* and *selection mechanism* interchangeably. In all further analysis, the three selection mechanisms are abbreviated as **TAP** (Selection-by-Tap), **GP** (Google Photos) and **MSA** (Multiselect-Android).

9.2 Metrics

We used the raw timestamp data described in Section 8.5, and extracted relevant data to help us measure the time and accuracy of user interactions. The raw data consisted of *events*. An *event* is a tuple $\langle type, timestamp \rangle$ where *type* indicates the selection command that was executed and *timestamp* the time at which the command was issued. Processing selection commands was fast (a few milliseconds), and hence we did not record the time at the end of processing an event. In MSA, dragging resulted in a rapid succession of double-tap events. Similarly in GP, dragging resulted

in a rapid succession of drag events. In these cases, we extracted the timestamp of the last repeating event in a series of events. The total time to execute a task was measured as the interval between the first and the last event before the user tapped the "Done" button. We ignored the time before the first event and the time after the last event. The time before the first event was the time taken by the user to survey the icons presented. The time after the last event was the time taken by the user to determine that the task was completed. Finally, we also recorded the number of gestures (taps, drags, double-taps, etc.) used by users in performing each task. In our measurement of gestures, we used the same method detailed above for coalescing successive double-tap and drag events.

9.3 Analysis

This section follows an organization similar to Section 8.4, starting with Task 0, and proceeding to results of the feedback acquired from users. We report the time taken and the number of gestures in each task. At each stage, we state our hypotheses, present results of basic statistical analyses, and then those of ANOVA tests with post-hoc analysis using the Tukey-Kramer method, where applicable, to check if the difference that we observed between selection mechanisms is statistically significant.

9.3.1 Task 0

Among the three mechanisms, TAP is the simplest and MSA the most complex, in terms of the number of different commands at the user's disposal. Consequently, the instructions of TAP are shorter than those of GP, which are in turn shorter than those of MSA. The user was encouraged to try each possible interaction, whose instructions were presented in a separate printed sheet. It is also likely that users had had prior experience with mechanisms similar to TAP and GP, which would

make them more familiar with these two mechanisms. We can thus expect that it takes a little more time to familiarize oneself with MSA than with GP or TAP.

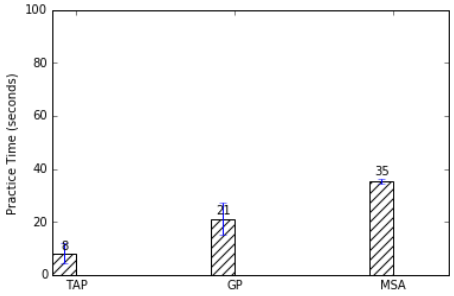
Hypothesis 0: MSA requires more practice time than either TAP or GP.

Figure 9.1 shows the average time and the average number of gestures used in the practice task. The error bars represent the standard errors. The single-factor ANOVA test revealed that the mechanism had a strong effect on practice time ($F_{2,33}$ = 10.87, p $\ll$ 0.001). The mechanism also had a strong effect on the number of gestures used ($F_{2,33}$ = 7.36, p $\ll$ 0.05). Tukey-Kramer post-hoc analyses revealed that there was a significant time difference between MSA and TAP techniques (p < 0.01), but not between MSA and GP. A similar analysis on gestures revealed a significant difference in the number of gestures used between GP and MSA (p < 0.01) and between TAP and MSA (p < 0.05). Interestingly, no conclusions could be drawn about the practice times under TAP and GP. This may be due to the inherent familiarity of the two mechanisms to the users. The analysis validates our hypothesis, and indicates that MSA may have a **steeper learning curve** than both TAP and GP. Nevertheless, the learning time was very short (less than a minute in all cases), and therefore it is unlikely to be a significant hurdle for the adoption and use of MSA.

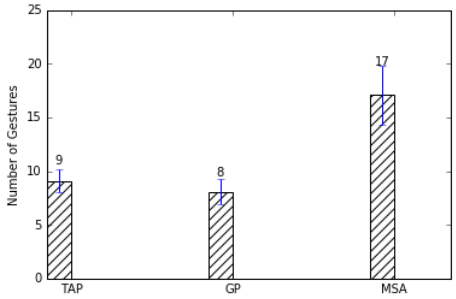
9.3.2 Task 1

All three mechanisms deal with selection and deselection of single elements in an identical manner. With each of them, the task completion time and the number of gestures in the simplest selection task involving selection of distinct, spatially disjoint elements is expected to be very similar.

Hypothesis 1: TAP, GP and MSA exhibit no significant difference when selecting disjoint elements.



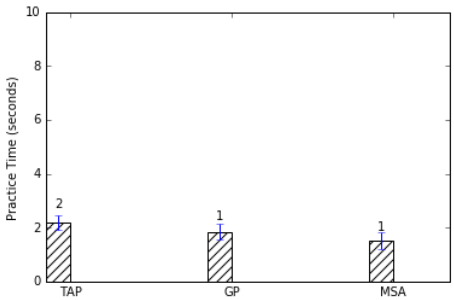
(a) Average time for each mechanism.



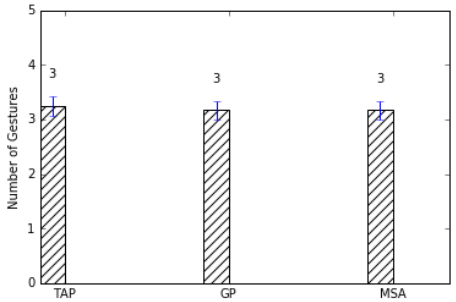
(b) Average number of gestures for each mechanism.

Figure 9.1: Statistics of Task 0 (practice).

Figure 9.2 shows the average time and the average number of gestures used in Task 1. There were almost no differences between the three mechanisms. It took on average about two seconds to complete the task using any of the mechanisms. The average number of gestures agree around three, which is expected for this task, where three elements had to be selected. There was no significant effect of the selection mechanism on task completion time ($F_{2,33} = 1.47$, NS), or on the number of gestures ($F_{2,33} = 0.08$, NS). Therefore, we can conclude in favor of our stated hypothesis. TAP, GP and MSA **perform identically when selecting distinct elements**.



(a) Average times for each mechanism.



(b) Average number of gestures for each mechanism.

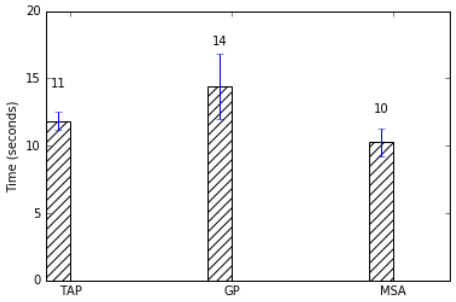
Figure 9.2: Statistics of Task 1.

9.3.3 Task 2

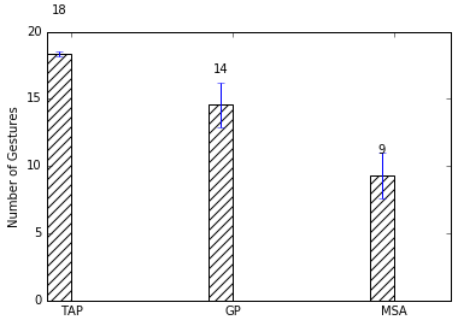
Task 2 required selecting several elements grouped spatially. Among the three selection mechanisms, TAP does not support selecting multiple elements using a single gesture, whereas GP and MSA provide such functionality. It was expected that this functionality would enable the user to perform this task faster with GP and MSA.

Hypothesis 2: MSA and GP outperform TAP in tasks requiring selection of contiguous groups of elements.

In our analysis, we discarded results with over 10% of the total number of elements incorrectly selected. This was because with many errors, it is possible to complete the task much quicker. Figure 9.3 shows the average time and the average number of gestures used in Task 2. The ANOVA test showed that the mechanism did not have a strong effect on task completion time ($F_{2,25} = 2.22$, NS) but did display a strong effect on the number of gestures ($F_{2,25} = 14.77$, $p \ll 0.01$). Post-hoc analysis revealed that both GP and MSA required significantly fewer gestures than TAP. Although the number of gestures was significantly higher in TAP, there was no significant difference in the time taken between TAP, GP and MSA. The reason may be that the contiguous groups of elements to be selected that we presented contained only 6–8 elements. These sizes may have been small enough that the, perhaps more familiar, tapping mechanism was competitive due to its simplicity. Hence, in this case, we accept the null hypothesis but note that there may have been a significant time difference between TAP and the other two mechanisms, had there been larger ranges to select as part of this task.



(a) Average times for each mechanism.



(b) Average number of gestures for each mechanism.

Figure 9.3: Statistics of Task 2.

9.3.4 Task 3

Task 3 consisted of a combination of distinct elements and contiguous groups of elements. The top half of the list of elements presented, including the viewport at the start contained a contiguous block of elements that had to be selected. It was followed by a set of elements scattered and already selected, which we refer to as *pre-selected elements*. These pre-selected elements had to remain selected at the end of the task in order to successfully complete the task. TAP was expected to be tedious with several selections to be performed. We also expected some users of TAP to quit the task early due to the tediousness. GP requires the user to hold a *drag* gesture near the edge of the viewport to scroll and simultaneously select multiple screens of elements in a single gesture. However, due to the imprecision of such scrolling, it was expected that the user would overshoot the contiguous unselected elements and cross over into the range of pre-selected scattered elements. Due to GP's *selection state erasing active domain*, backtracking within the same drag gesture would cause the pre-selected elements to become unselected, requiring another set of taps from the user to select them again and complete the task.

MSA allows overlapping selections to remain independent as it has a *selection state preserving active domain*. MSA also offers the ability to select and deselect ranges of elements which may span multiple screens over several gestures interspersed with scrolling, rather than a single drag gesture with its limited ability to control scrolling. MSA is not affected by the issues we detailed above with regard to GP. Due to this, we expected MSA to perform the best, followed by GP and finally TAP.

Hypothesis 3.1: TAP performs worse than GP and MSA when presented with large contiguous groups of selections.

Hypothesis 3.2: MSA performs better than GP when presented with large contiguous groups of elements spanning multiple screens to be selected, in the presence of previous selections immediately following them.

In our analysis, we discarded records with over 10% of the elements incorrectly selected. The TAP data-set was considerably smaller than the others due to large number of errors. This is presumably due to users giving up before completing all selections as expected. Figure 9.4 shows the average time and the average number of gestures observed in Task 3. A single-factor ANOVA test revealed that the mechanism had a strong effect on the time to perform task 3 ($F_{2,26} = 49.71$, $p \ll 0.001$), and also on the number of gestures ($F_{2,26} = 36.29$, $p \ll 0.001$). Tukey-Kramer post-hoc analyses revealed that of the three pairs, there existed a significant difference between all three pairs of techniques, shown in Table 9.1 and 9.2. The difference between MSA and GP can be attributed to the selection state preserving nature of MSA described above.

Both our hypotheses were confirmed and for this particular arrangement, MSA outperforms GP, which in turn performs better than TAP.

Task 3 may seem a little contrived; one might expect that the user would select items in order, not starting from the bottom, jumping to the top, and moving back

	p-value	**Q Statistic**
GP v/s MSA	p<0.01	4.5674
GP v/s TAP	p<0.05	3.6950
MSA v/s TAP	p<0.01	7.8268

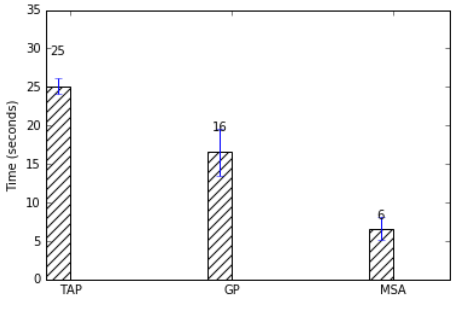
Table 9.1: Results of Tukey-Kramer test for the time statistic in Task 3.

	p-value	**Q Statistic**
GP v/s MSA	p<0.05	4.141
GP v/s TAP	p<0.01	8.2625
MSA v/s TAP	p<0.01	11.9083

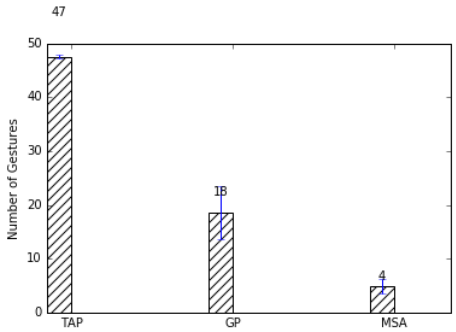
Table 9.2: Results of the Tukey-Kramer test for the gesture statistic in Task 3.

towards the already selected elements at the bottom. We argue, however, that a good multi-selection feature should allow a complete freedom in which order to select elements. The above kind of selection order might appear in practice, for example, with typical email clients that present the most recent messages at the bottom. The user could select a few messages for deletion at the bottom, then decide that a larger block of earlier emails in the same discussion should be selected as well. Another example could be views where elements can be reordered during the selection. For example, in Apple's Finder, one can select a block of files, then reorder the view, e.g., based on file size. This might result in distinct selected items.

We conjecture that a selection feature where the *active domain* is selection preserving is superior to one where it is selection erasing. We also conjecture that a selection feature that allows one to perform a partial range selection, scroll in any means the user prefers, and then continue the range selection, is superior to one where a drag must be performed in one gesture (scrolling must in this latter case be perfomed by pushing against a screen edge). The results of task 3 can be interpreted



(a) Average times for each mechanism.



(b) Average number of gestures for each mechanism.

Figure 9.4: Statistics of Task 3.

to support these conjectures.

9.3.5 Task 4

In this task, the goal was to test the effect of deselection on completion time of the task. TAP was expected to be worse than the other two alternatives. MSA was expected to be better than GP, because MSA features a deselection mode which could be used to perform the task in fewer steps. The abilility to perform deselections on a range of elements is absent in GP. The reason the rectangular geometry was chosen for this task is that the difference in the minimum number of gestures that could be used to complete a task with and without the ability to deselect ranges of elements is larger.

Hypothesis 4: MSA outperforms GP when the ability to deselect ranges of elements allows the use of fewer gestures to carry out selections.

In Task 4 there were very few errors (<10% in all cases), enabling us to include all results. Figure 9.5 shows the average time and the average number of gestures used. The single-factor ANOVA test revealed that the mechanism did have an effect on the time of this task ($F_{2,33} = 6.88$, $p < 0.01$) and on the number of gestures

($F_{2,33} = 43.17$, p $\ll 0.001$). Tukey-Kramer post-hoc analysis (Table 9.3) revealed that there was a significant difference between the task completion times of GP and MSA. Similar analysis (Table 9.4) revealed no significant difference in the number of gestures between the GP and MSA.

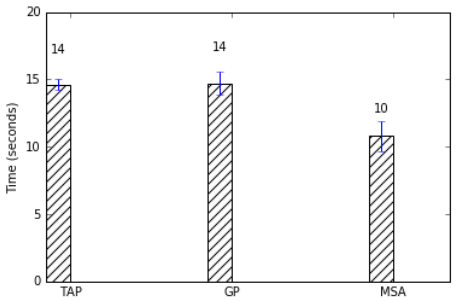
Although the users completed the selection task significantly faster with MSA, they did not use fewer gestures than their counterparts using GP. This appears to indicate that users may not have used the deselection gestures to accelerate the task. The time difference observed may have come from greater flexibility in gestures for selection of ranges of elements offered by MSA. However, the utility of deselections could not be confirmed and we reject the hypothesis.

	p-value	**Q Statistic**
GP v/s MSA	p<0.01	4.6011
GP v/s TAP	ns	0.1407
MSA v/s TAP	p<0.01	4.4605

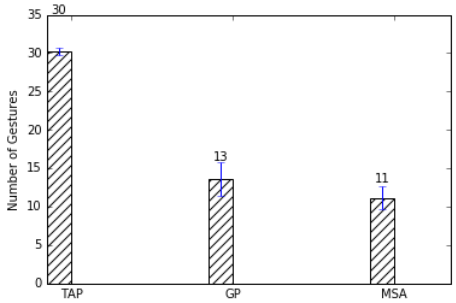
Table 9.3: Results of the Tukey-Kramer test for the time statistic in Task 4.

	p-value	**Q Statistic**
GP v/s MSA	ns	1.5839
GP v/s TAP	p<0.01	10.5064
MSA v/s TAP	p<0.01	12.0902

Table 9.4: Results of the Tukey-Kramer test for the gestures statistic in Task 4.



(a) Average times for each mechanism.



(b) Average number of gestures for each mechanism.

Figure 9.5: Statistics of Task 4.

9.3.6 User feedback

We analyzed the user feedback we acquired for the two questions posed to the users after they had completed all four tasks. Some users volunteered to provide additional feedback and remarked about the lack of an ability to deselect multiple elements with a single gesture in GP. Several users also mentioned that TAP was tedious for the tasks presented.

As shown in Figure 9.6, TAP was reported to be insufficient, and frustrating by several users and was significantly worse than MSA or GP. This was expected as TAP does not offer any gestures to select many elements and several tasks were rather tedious to perform without such a facility. Most users reported both MSA and GP to be sufficient to perform the tasks, but fewer users found MSA frustrating.

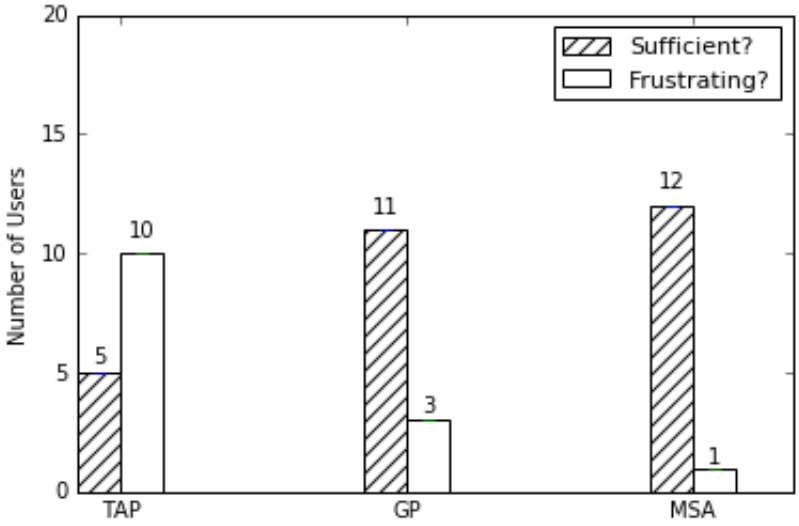


Figure 9.6: Feedback reported by users at the end of all tasks

10. FUTURE WORK

In this section, we outline some of the future work that we plan to undertake in this domain. There are several areas of research closely related to this thesis that we would like to develop further.

We would like to research space optimizations, that may be made possible by our semantics, when dealing with large numbers of selectable elements. A large number of selectables may exist when each selectable is very small, say, when each pixel in an image can be selected. Space optimizations may be in the form of an alternative representation of selection mappings.

We would also like to investigate various optimizations for improving the running time of various selection commands in practice. Modern mobile computing devices typically feature multiple processing cores and an interesting direction would be to exploit these multi-core architectures and utilize parallelism in the computation of selection domains and selections.

We are also interested in applying our semantics to other areas not typically thought of as multi-selection, such as text selection. A word processor may define taps of varying multiplicities interpreted to be interpreted as different types of selections. For example, a single-tap could select a single character, a double-tap, a word, and a triple-tap, an entire paragraph. We are interested in coming up with selection geometries or modifications that allows us to incorporate these unconventional selection models in our semantics.

11. CONCLUSION

In this thesis we have presented an in-depth analysis of multi-selection and a formal definition of its semantics for touch-screen interfaces. We have also presented an implementation of our semantics in the form of a library, *Multiselect-Android*, and a user study we carried out to test the effectiveness of our semantics. Our conclusions are as follows:

- The formalism we presented precisely describes multi-selection for touch-screen interfaces as a reusable feature.
- From the comparison of our implementation with rival multi-selection frameworks, we conclude that Multiselect-Android makes the implementation of multi-selection considerably less difficult, less error-prone, and more flexible by allowing the implementer to focus only on the application-specific aspects of multi-selection.
- The results of our user study have demonstrated that Multiselect-Android is no worse than rival selection mechanisms in carrying out simple selection tasks, and outperforms rival implementations by a significant margin when presented with complex selection tasks. User feedback showed that our technique was sufficient to perform a wide range of selection tasks while being the least frustrating to use among two other mechanisms.

REFERENCES

[1] Aidan Follestad. *The drag-select-recyclerview library*. URL: https://github.com/afollestad/drag-select-recyclerview (visited on 03/28/2016).

[2] Apple Computer, Inc. *Macintosh Human Interface Guidelines*. USA: Addison-Wesley, 1992. ISBN: 0-201-62216-5.

[3] Apple Inc. *Selections in UITableView*. URL: https://developer.apple.com/library/ios/samplecode/TableMultiSelect/Introduction/Intro.html (visited on 03/28/2016).

[4] Stephen J Bisset and Bernard Kasser. *Multiple fingers contact sensing method for emulating mouse buttons and mouse operations on a touch sensor pad*. US Patent 5,825,352. 1998.

[5] Wen-hsiang Chiang. *Multi-touch electronic device, graphic display interface thereof and object selection method of multi-touch display*. US Patent App. 12/987,577. 2011.

[6] H. Dehmeshki and W. Stuerzlinger. "Design and evaluation of a perceptual-based object group selection technique". In: *Proceedings of the 24th BCS Interaction Specialist Group Conference*. British Computer Society. 2010, pp. 365–373.

[7] Google Inc. *Google Photos*. URL: https://play.google.com/store/apps/details?id=com.google.android.apps.photos&hl=en (visited on 03/28/2016).

[8] Google Inc. *Material Design Guidelines for Item and Text Selection*. URL: https://www.google.com/design/spec/patterns/selection.html (visited on 03/28/2016).

[9] Tor-Morten Grønli et al. "Mobile application platform heterogeneity: Android vs Windows Phone vs iOS vs Firefox OS". In: *Advanced Information Networking and Applications (AINA), 2014 IEEE 28th International Conference on*. IEEE. 2014, pp. 635–641.

[10] Jaakko Järvi and Sean Parent. *MultiselectJS*. 2016. URL: http://hotdrink.github.io/multiselectjs/.

[11] Jaakko Järvi and Sean Parent. "One Way to Select Many". In: *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. To appear. 2016.

[12] Jonathan Lazar et al. "Severity and Impact of Computer User Frustration: A Comparison of Student and Workplace Users". In: *Interact. Comput.* 18.2 (Mar. 2006), pp. 187–207. ISSN: 0953-5438. DOI: 10.1016/j.intcom.2005.06.001. URL: http://dx.doi.org/10.1016/j.intcom.2005.06.001.

[13] Jakob Leitner and Michael Haller. "Harpoon Selection: Efficient Selections for Ungrouped Content on Large Pen-based Surfaces". In: *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*. UIST '11. Santa Barbara, California, USA: ACM, 2011, pp. 593–602. ISBN: 978-1-4503-0716-1. DOI: 10.1145/2047196.2047275. URL: http://doi.acm.org/10.1145/2047196.2047275.

[14] Microsoft. *Windows Interface Guidelines for Software Design*. 1st. Redmond, WA, USA: Microsoft Press, 1995. ISBN: 1556156790.

[15] Sachi Mizobuchi and Michiaki Yasumura. “Tapping vs. Circling Selections on Pen-based Devices: Evidence for Different Performance-shaping Factors”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’04. Vienna, Austria: ACM, 2004, pp. 607–614. ISBN: 1-58113-702-8. DOI: 10.1145/985692.985769. URL: http://doi.acm.org/10.1145/985692.985769.

[16] Donald A. Norman and Jakob Nielsen. “Gestural Interfaces: A Step Backward in Usability”. In: *interactions* 17.5 (Sept. 2010), pp. 46–49. ISSN: 1072-5520. DOI: 10.1145/1836216.1836228. URL: http://doi.acm.org/10.1145/1836216.1836228.

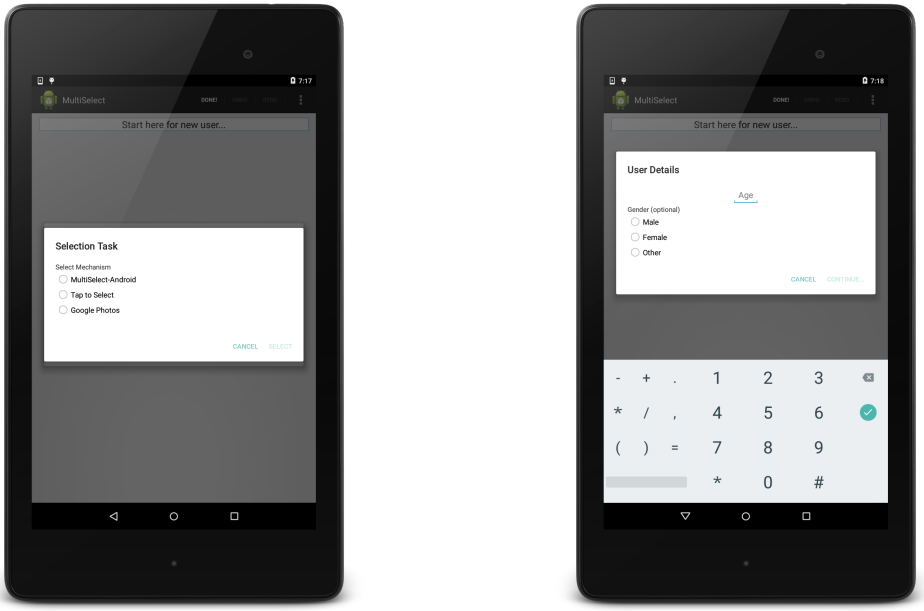
[17] Ofri Olavi Porat. *Apparatus, method and computer program product for selecting multiple items using multi-touch*. US Patent App. 12/118,975. 2008.

[18] Volker Roth and Thea Turner. “Bezel Swipe: Conflict-free Scrolling and Multiple Selection on Mobile Touch Screen Devices”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’09. Boston, MA, USA: ACM, 2009, pp. 1523–1526. ISBN: 978-1-60558-246-7. DOI: 10.1145/1518701.1518933. URL: http://doi.acm.org/10.1145/1518701.1518933.

APPENDIX A

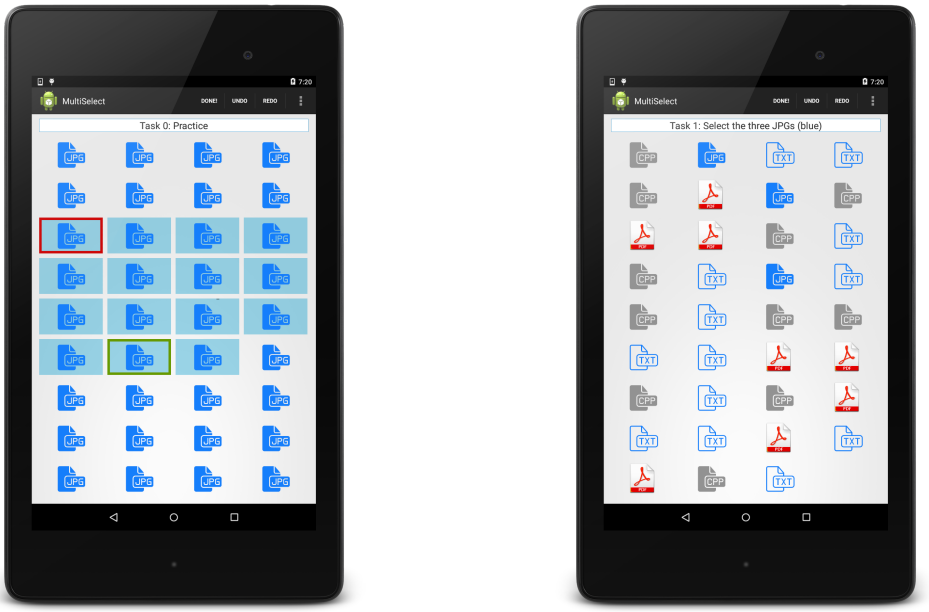
APPLICATION USED IN OUR USER STUDY

A.1 Screen Captures



(a) Mechanism selection. (b) Age and gender input.

Figure A.1: Initial screens for mechanism selection and user input before the start of user study tasks. The mechanism selection screen is not exposed to users.



(a) Task 0.

(b) Task 1.

Figure A.2: Tasks 0 and 1

(a) Task 2.

(b) Task 3.

Figure A.3: Tasks 2 and 3

(a) Task 4. (b) Feedback.

Figure A.4: Task 4 and Feedback.